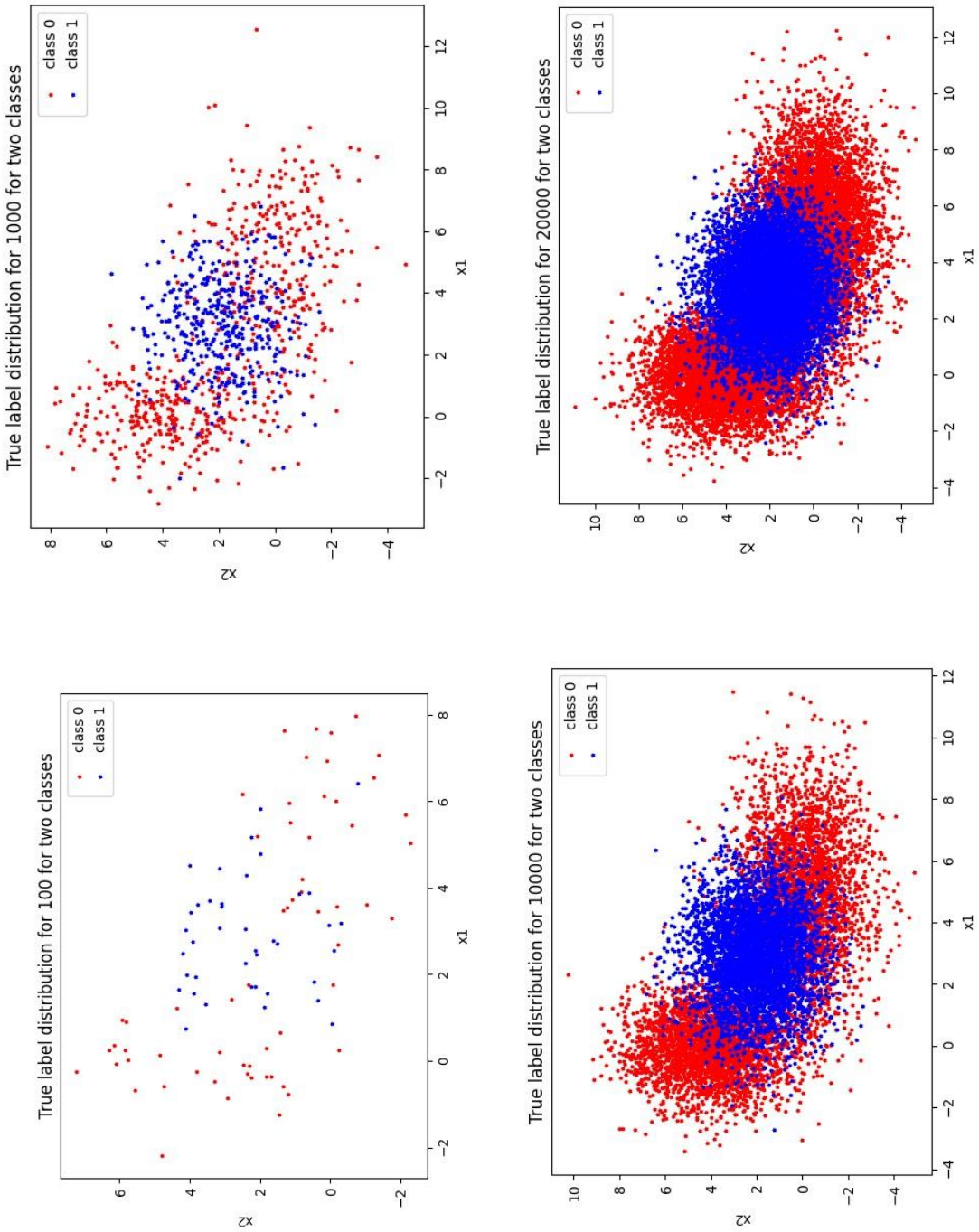
Q1. (Code in Appendix)

Plots for the datasets used are shown below:



- D^{100}, D^{1000}, D^{10000}, D^{20000} $_{validate}$ consists of 100, 1000, 10000, 20000 data samples and their labels for validation respectively.

Goutham Saravanan
EECE5644 – HW3

A) Theoretically Optimal Classifier with Known Parameters

The theoretically optimal classifier was determined using knowledge about the true pdf in this section. A likelihood-ratio test as a minimum expected risk classification rule is:

$$\frac{P(X|L=1)}{P(X|L=0)} \underset{D(x)=0}{\overset{D(x)=1}{\gtrless}} \frac{(\lambda_{10}-\lambda_{00})P(L=0)}{(\lambda_{01}-\lambda_{11})P(L=1)}$$

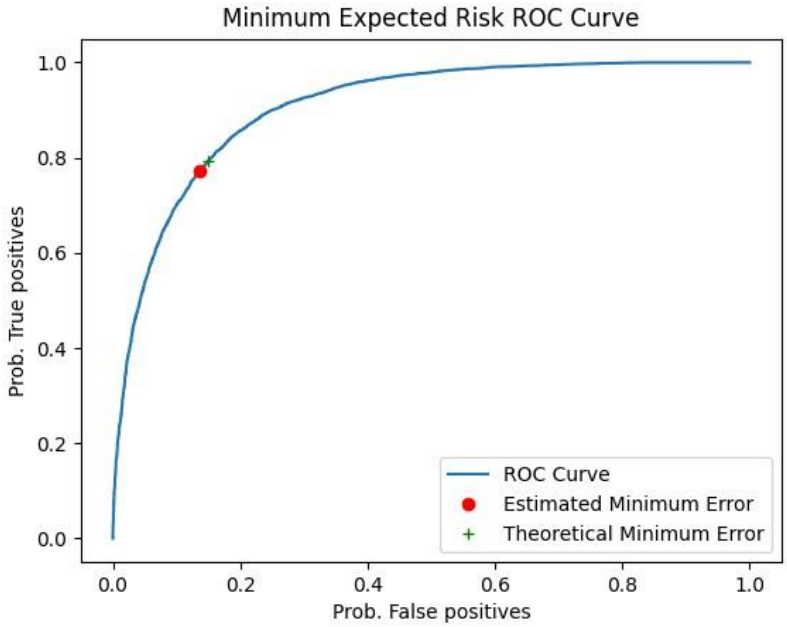
The cost of wrong classification should be 1 and the cost of accurate classification should be 0. This will lower the likelihood of misclassifications. The likelihood-ratio test for this situation is presented below.

$$\frac{P(X|L=1)}{P(X|L=0)} \underset{D(x)=0}{\overset{D(x)=1}{\gtrless}} \frac{(1-0)*0.6}{(1-0)*0.4} = 1.5 = \gamma$$

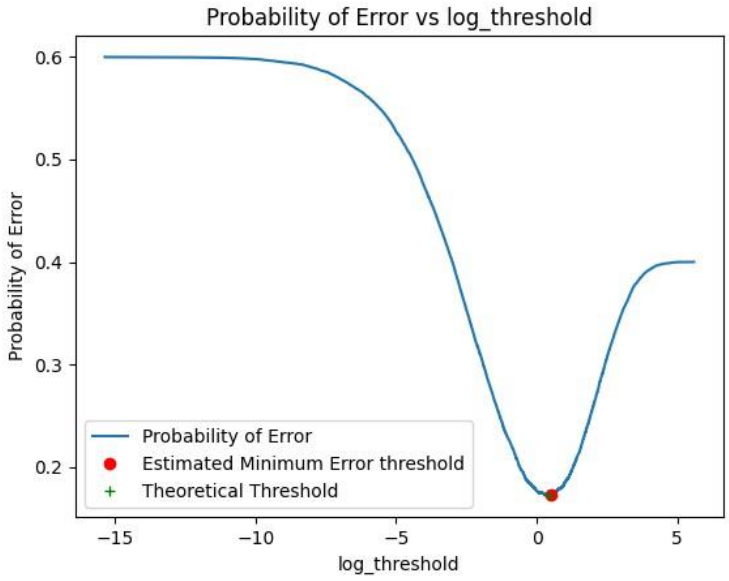
For a set of threshold values, a parametric sweep was performed, and a classifier was found for each of these threshold values. For each of these threshold values, true positives and false positives were determined, and the ROC curve was shown. Below is a diagram of the plot.

	Threshold value (γ)	Minimum probability of Error
Theoretical	1.5	17.28%
Estimation from the data sample	1.68	17.22%

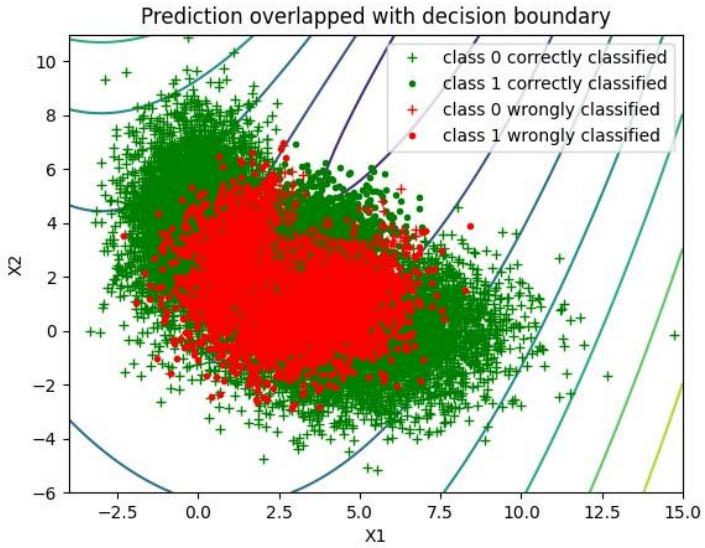
Goutham Saravanan
EECE5644 – HW3



- The theoretically optimal threshold value's minimum probability of error (min PE) was determined, and the operating point in red colour plotted of this error was superimposed on the ROC curve above.



Goutham Saravanan
EECE5644 – HW3



- The decision boundary for each distribution is shown above, along with the discriminant function's equi-level contours.

B) Classifier with Estimated Parameters

For this section, classification was done based on best-guess understanding of the data's underlying distributions. Class 0 was modeled as a two-component Gaussian Mixture Model, whereas Class 1 was modeled as a single Gaussian. Each of the three training datasets having 100, 1000, and 10000 samples was used to estimate parameters, which were then used to classify a dataset containing 20,000 samples.

The likelihood function for a Gaussian Model is defined as

$$\begin{aligned}\mathcal{L} = p(\mathbf{X}\,|\,\theta) &= \mathcal{N}(\mathbf{X}\,|\,\theta) \\ &= \mathcal{N}(\mathbf{X}\,|\,\mu, \Sigma)\end{aligned}$$

where X is the dataset, μ, Σ are the mean and covariance of the distribution.

- To get a good MLE of model, we need good estimates of mean and covariance. They can be calculated as follows,

$$\mu_{MLE} = \text{argmax}_{\mu}\mathcal{N}(\mathbf{X}\,|\,\mu, \Sigma)$$
$$\Sigma_{MLE} = \text{argmax}_{\Sigma}\mathcal{N}(\mathbf{X}\,|\,\mu, \Sigma)$$

- In Python, The built-in function sklearn.mixture.GuassianMixture was used to estimate parameters for Class 0 and Class 1. The Expectation-Maximization (EM) algorithm is the iterative numerical optimization approach utilized here. As demonstrated below, this technique optimizes the anticipated value of the log likelihood function.

$$Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)}) = \mathrm{E}_{\mathbf{Z}|\mathbf{X},\boldsymbol{\theta}^{(t)}}\left[\log L(\boldsymbol{\theta}; \mathbf{X}, \mathbf{Z})\right]$$

$$\boldsymbol{\theta}^{(t+1)} = \arg\max_{\boldsymbol{\theta}} Q(\boldsymbol{\theta} \mid \boldsymbol{\theta}^{(t)})$$

  - The estimated means, covariance, and weights obtained after training on 100 data samples are shown below.

$$m_{01} = \begin{bmatrix}5.09\\0.12\end{bmatrix} \quad C_{01} = \begin{bmatrix}3.17 & 0.39\\0.39 & 1.27\end{bmatrix} \quad m_{02} = \begin{bmatrix}-0.13\\4.13\end{bmatrix} \quad C_{02} = \begin{bmatrix}0.87 & 0.06\\0.06 & 2.33\end{bmatrix}$$

$$m_1 = \begin{bmatrix}3.15\\1.85\end{bmatrix} \quad C_1 = \begin{bmatrix}1.98 & 0.04\\0.04 & 0.99\end{bmatrix} \quad w1 = 0.47 \; w2 = 0.52$$

- For 1000 samples:

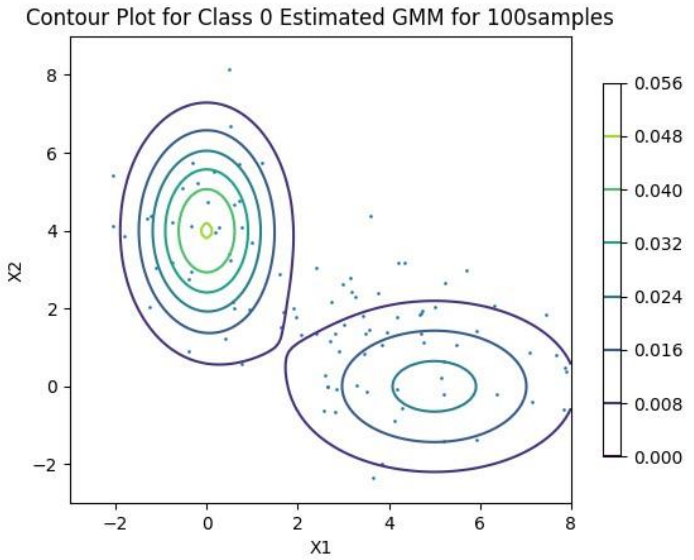
$$m_{01} = \begin{bmatrix} 5.1 \\ -0.17 \end{bmatrix} \; C_{01} = \begin{bmatrix} 3.87 & 0.18 \\ 0.18 & 2.11 \end{bmatrix} m_{02} = \begin{bmatrix} 0.1 \\ 3.79 \end{bmatrix} \; C_{02} = \begin{bmatrix} 1.1 & -0.08 \\ -0.08 & 3.04 \end{bmatrix}$$
$$m_1 = \begin{bmatrix} 3.01 \\ 2.1 \end{bmatrix} \; C_1 = \begin{bmatrix} 2.18 & -0.07 \\ -0.07 & 1.9 \end{bmatrix} \; w1 = 0.50 \; w2 = 0.49$$

- For 10000 samples:.

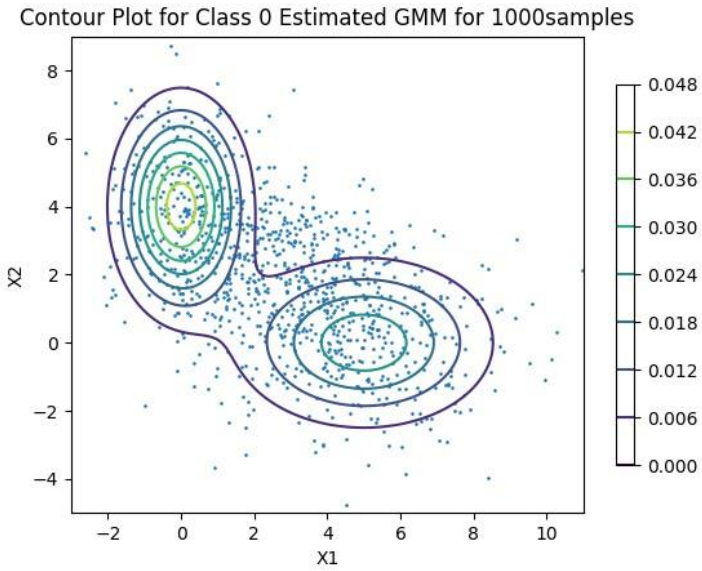
$$m_{01} = \begin{bmatrix} 5.0 \\ -0.01 \end{bmatrix} \; C_{01} = \begin{bmatrix} 3.88 & 0.01 \\ 0.01 & 1.99 \end{bmatrix} m_{02} = \begin{bmatrix} 0.0 \\ 3.9 \end{bmatrix} \; C_{02} = \begin{bmatrix} 0.99 & -0.01 \\ -0.01 & 3.07 \end{bmatrix}$$
$$m_1 = \begin{bmatrix} 3.01 \\ 1.98 \end{bmatrix} \; C_1 = \begin{bmatrix} 2.07 & 0.00 \\ 0.00 & 2.02 \end{bmatrix} \; w1 = 0.50 \; w2 = 0.49$$

- Below table shows the sample class priors for each of the training data.

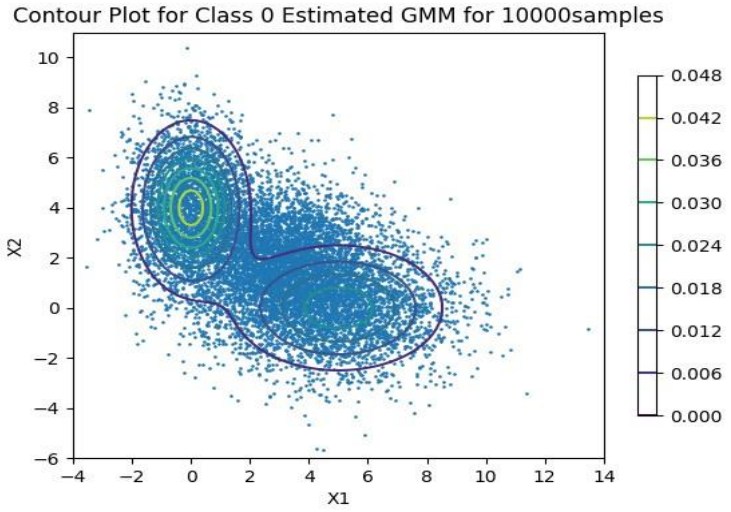
	$D_{100_{train}}$	$D_{1000_{train}}$	$D_{10000_{train}}$
P(L=0)	0.66	0.63	0.61
P(L=1)	0.34	0.37	0.39



- For 100 training samples, the contour of estimated distributions for Class 0 Gaussian Mixture Model(GMM) is shown above.



- For 1000 samples GMM model



- GMM model for 10000 samples.

- Below is a summary of the minimum estimated likelihood of errors for the parameters estimated from the three training datasets and validated on 20,000 data samples.

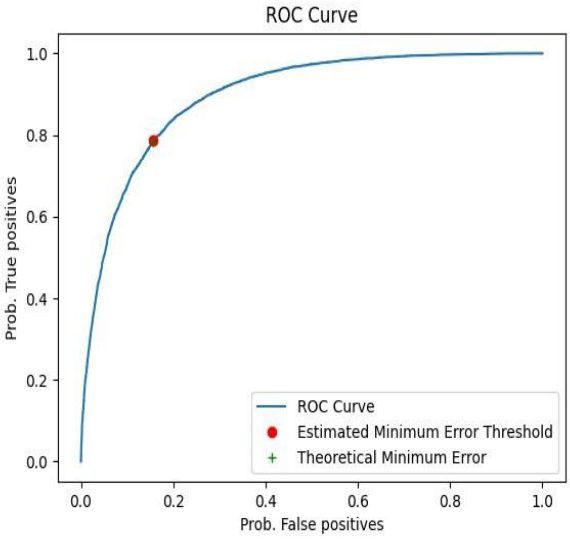
Training samples	Min Error threshold value (γ)	Probability of Error
100	1.46	17.9%
1000	1.26	17.6%
10000	1.61	17.5%
Known PDF(Part 1)	1.68	17.2%

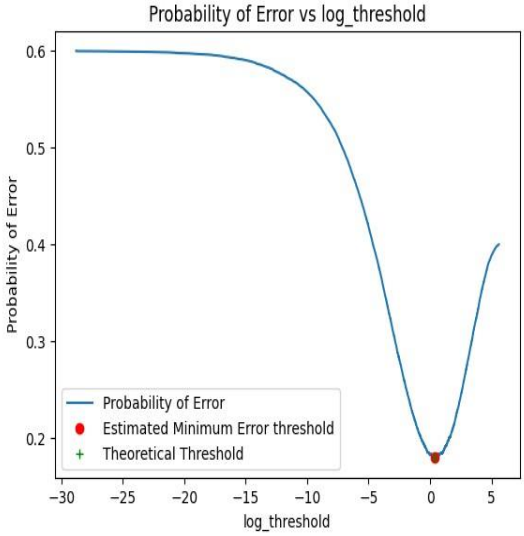
Observation:

- The parameter estimations (mean, covariance, and weights) become closer to the real value as the number of training samples grows.
- The chance of error on the validation dataset decreases as the number of training samples increases.

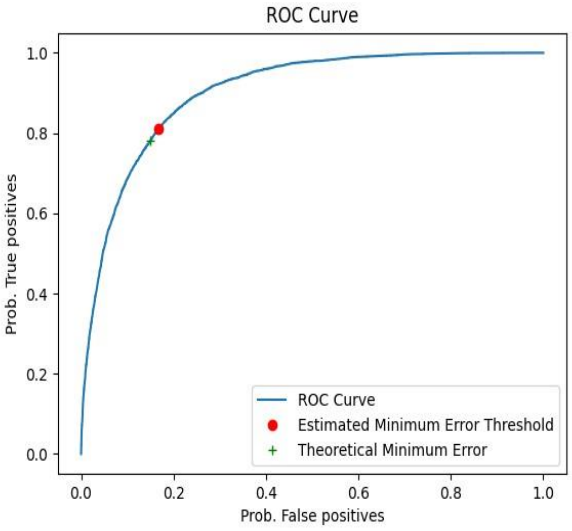
- A model trained on 10,000 samples has a probability of error that is similar to the error computed using known data (part 1).

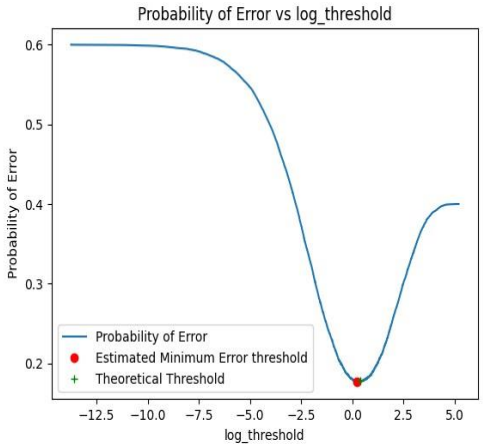
- ROC Curve and Minimum Probability of Error plot for training data: **D100** validate data: **D20000**



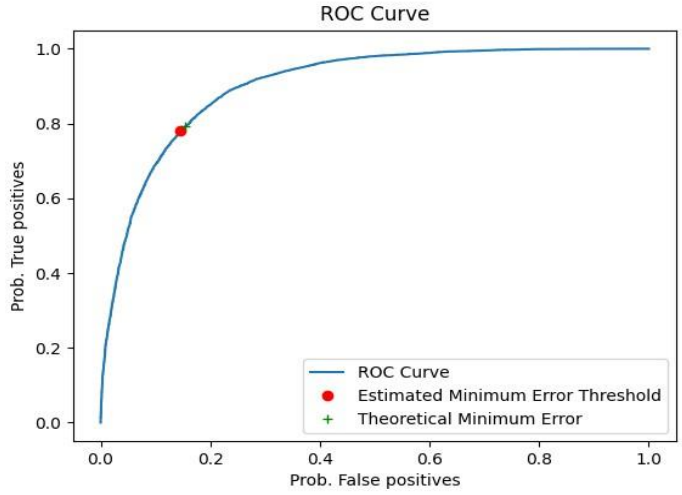


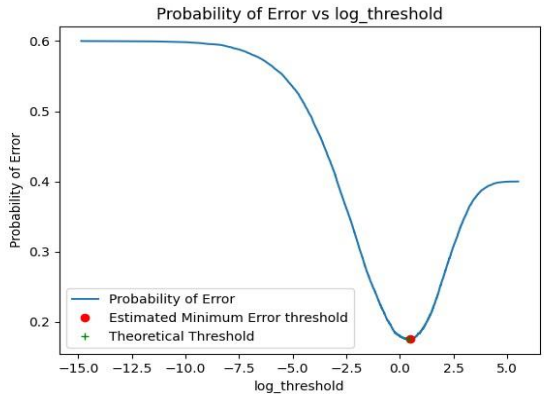
- ROC Curve and Minimum Probability of Error plot for training data: **D1000** validate data: **D20000**





- ROC Curve and Minimum Probability of Error plot for training data: **D10000** validate data: **D20000**





C) Classifier using Logistic Function:

1. On a given dataset, maximum likelihood parameter estimation approaches were utilized to train logistic linear and logistic quadratic-based approximations of class label posterior functions. Validation was performed on a dataset comprising 20,000 samples after training on three independent datasets having 100, 1000, and 10000 samples.

2. The logistic function is shown below:

$$h(x, w) = \frac{1}{1+e^{w^T z(x)}}$$

3. For linear logistic function $z(x) = [1\ x_1\ x_2]^T$
4. For quadratic logistic function $z(x) = [1\ x_1\ x_2\ x_1^2\ x_1x_2\ x_2^2]^T$
5. The built-in function *scipy.optimize.minimize* was used for numerical optimization of both the functions.

6. The learnable vector(w) is estimated using numerical optimization techniques with the cost function as shown below

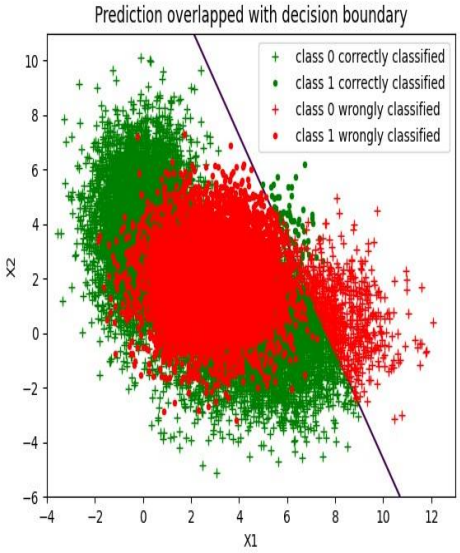
$$\hat{\theta}_{ML} = -\frac{1}{N}\sum_{1}^{N} l_n \ln\big(h(x_n, \theta)\big) + (1 - l_n)\ln\big(1 - h(x_n, \theta)\big)$$

7. The minimum expected risk classification criteria is
   i. $(l_n = 1) \qquad \hat{w}^T z(x) \gtrless 0 \qquad (l_n = 0)$
8. The probability of error (PE) for classifiers trained on the three datasets and evaluated on a dataset with 20,000 samples for both linear and quadratic logistic fit is summarized in the table below. The likelihood of inaccuracy lowers as the number of training samples increased. The reduction in the likelihood of error is less because classifiers are restricted by the approximation capabilities of their functional form. The probability of mistake in a quadratic logistic function is substantially lower than in a linear logistic function due to the increased complexity. This function approximated the theoretical optimal probability of error of 17.2% after being trained on 10,000 data in part1.

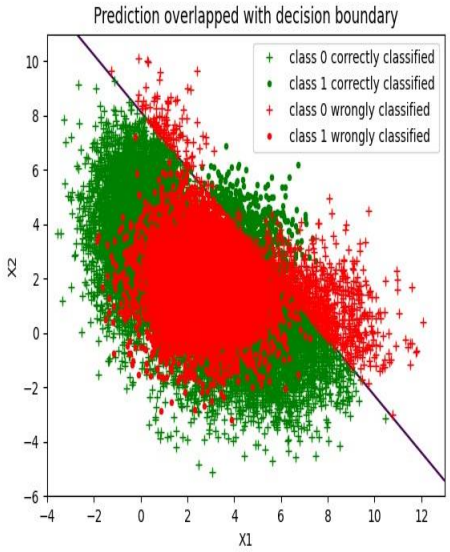
	PE $D_{100train}$	PE $D_{1000train}$	PE $D_{10000train}$
Linear Logistic Fit	42.7%	42.6%	41.2%
Quadratic Logistic Fit	20.0%	17.4%	17.2%

Data and Classifier Decision on True Label for Linear Logistic Fit

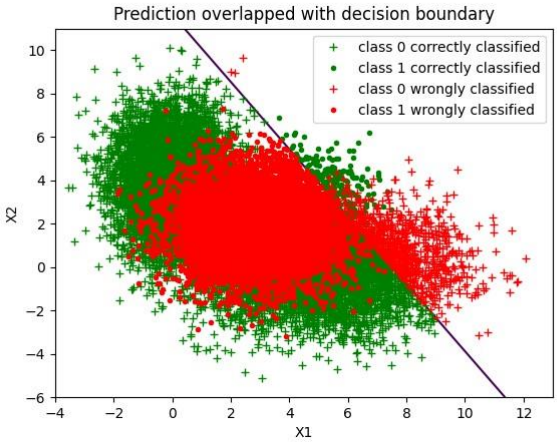
Train Data: 100 Validate Data: 20000
Probability of Error = 42.7%



Train Data: 1000 Validate Data: 20000
Probability of Error = 42.6%

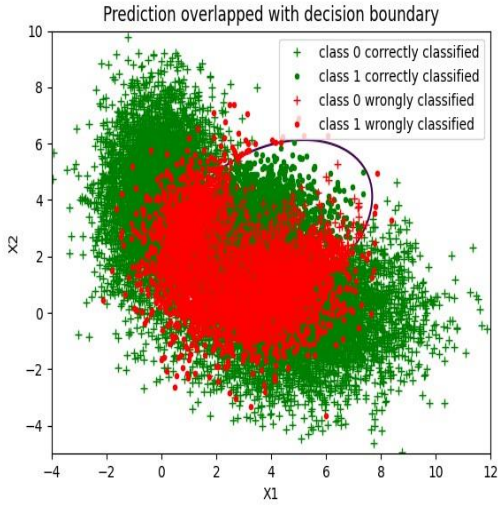


Train Sample Data: 10000 Validate Data: 20000
Probability of Error = 41.2%

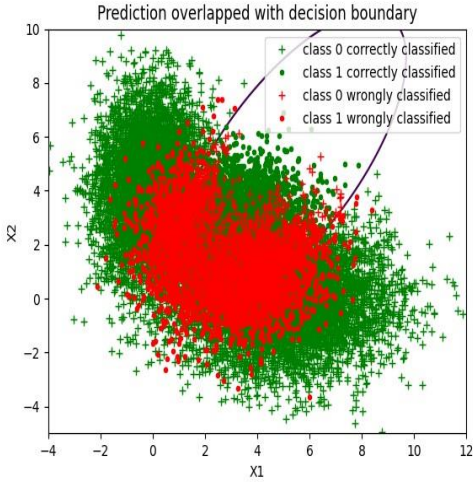


Data and Classifier Decision on True Label for Quadratic Logistic Fit

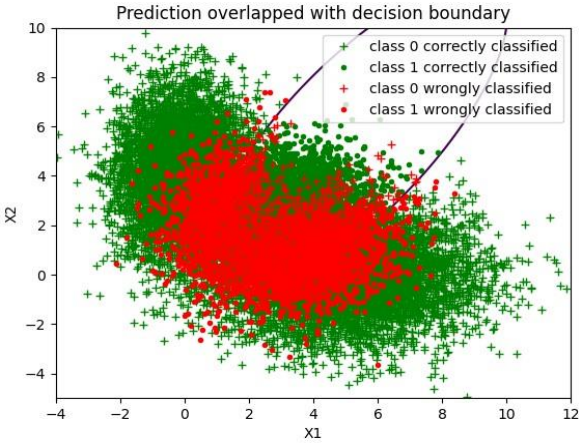
Sample Training Data: 100 Validate Data: 20000
Probability of Error = 20.0%



Sample Training Data: 1000 Validate Data: 20000
Probability of Error = 17.4%



Sample Train Data: 10000 Validate Data: 20000
Probability of Error = 17.2%



Q2:

Answer: A velicle at the position $[x_T, y_T]^T$.

& its' respective co-ordinates $[x_1, y_1]^T, [x_2, y_2]^T \ldots . [x_k, y_k]^T$.

Range : $r = d_{Ti} + n_i$, $i \in \{1, \ldots k\}$.

So, $p[x, y]^T = (2\pi \sigma_x \sigma_y)^{-1} \cdot e^{-\frac{1}{2}[x\ y]\begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix}}$

(A) - Also, given that $d_{Ti} = \left\| \begin{bmatrix} x_T \\ y_T \end{bmatrix} - \begin{bmatrix} x_i \\ y_i \end{bmatrix} \right\|$. & $n_i \sim N(0, \sigma_i^2)$

From (A) we could say.

$$r_i \sim N(d_{Ti}, \sigma_i^2).$$

Now, to determine the MAP estimate of the velicle position is

$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\arg\max}\ p\left(\begin{bmatrix} x \\ y \end{bmatrix} | r\right) \quad \text{(B)}$$

Now applying Bayes Theorem, to (B) -

$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\text{argmax}} \; P\left(r \middle| \begin{bmatrix} x \\ y \end{bmatrix}\right) \cdot P\left(\begin{bmatrix} x \\ y \end{bmatrix}\right)$$

$$= \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\text{argmax}} \left(\prod_{i}^{k} P\left(r_i \middle| \begin{bmatrix} x \\ y \end{bmatrix}\right) \right) \cdot P\begin{bmatrix} x \\ y \end{bmatrix} \quad \text{— (C)}$$

Now introducing the pdf function to (C).

$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\text{argmax}} \left(\prod_{i}^{k} (2\pi)^{-n/2} |\sigma_i^2)^{-1/2} \cdot e^{-1/2 \frac{(r_i - d_{Ti})^2}{\sigma_i^2}} \right) \times$$

$$\left((2\pi \sigma_x \sigma_y)^{-1} \cdot e^{-\frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{pmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{pmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix}}\right) .$$

(D)

Now, applying ln (log) function to the R.H.S.

$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\text{argmax}} \left(\sum_{1}^{k} -\frac{n}{2} \ln(2\pi) - \frac{1}{2} \ln(\sigma_i^2) - \frac{1}{2} \frac{(r_i - d_{Ti})^2}{\sigma_i^2} \right)$$

$$- \ln(2\pi \sigma_x \sigma_y) - \frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix} - \text{(E)}$$

From (E), after applying the log fn the constant could be avoid as it does not affect the maximum point.

So,

$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} \sum_{1}^{K} -\frac{1}{2} \frac{(r_i - d_{Ti})^2}{\sigma_i^2} - \frac{1}{2} \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix}$$

Arranging the common terms,

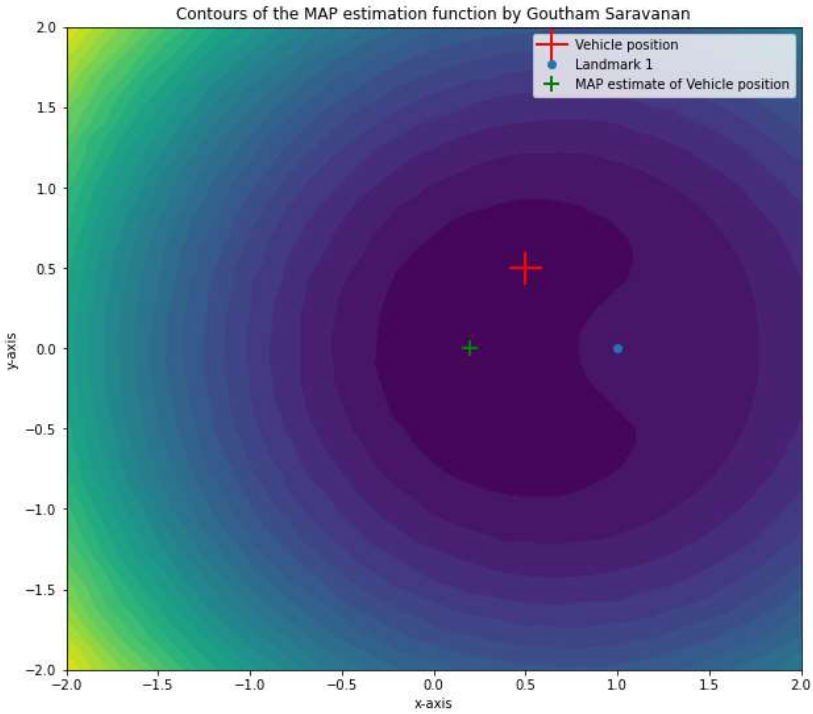
$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} \sum_{1}^{K} \frac{(r_i - d_{Ti})^2}{\sigma_i^2} + \begin{bmatrix} x \\ y \end{bmatrix}^T \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \end{bmatrix}. \quad \text{(F)}$$

Multiplying the matrix inside the eq. (F).

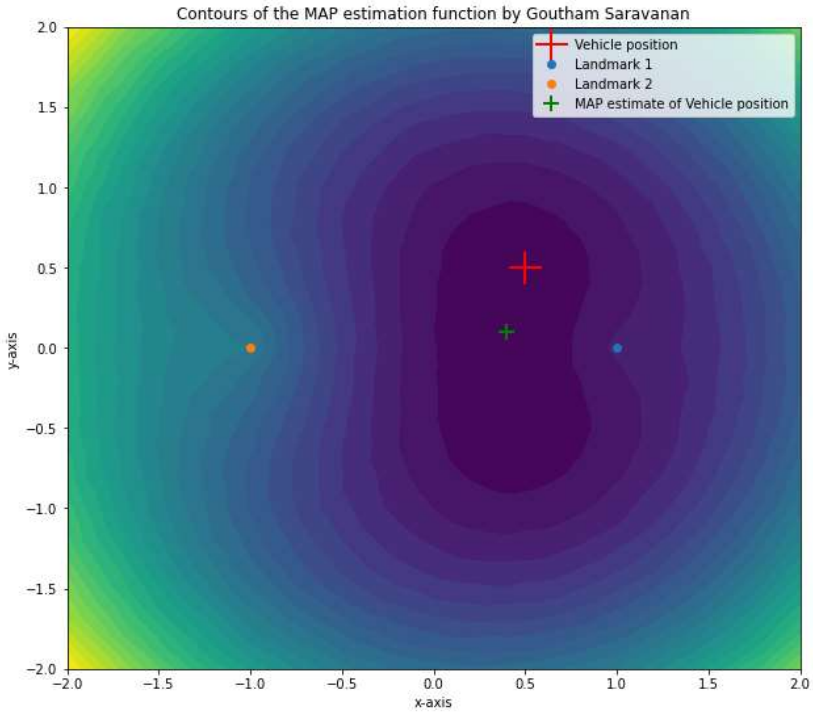
So, we have

$$\begin{bmatrix} x \\ y \end{bmatrix}_{MAP} = \underset{\begin{bmatrix} x \\ y \end{bmatrix}}{\operatorname{argmax}} \sum_{1}^{K} \frac{(r_i - d_{Ti})^2}{\sigma_i^2} + \frac{x^2}{\sigma_x^2} + \frac{y^2}{\sigma_y^2}. \quad \text{(G)}$$

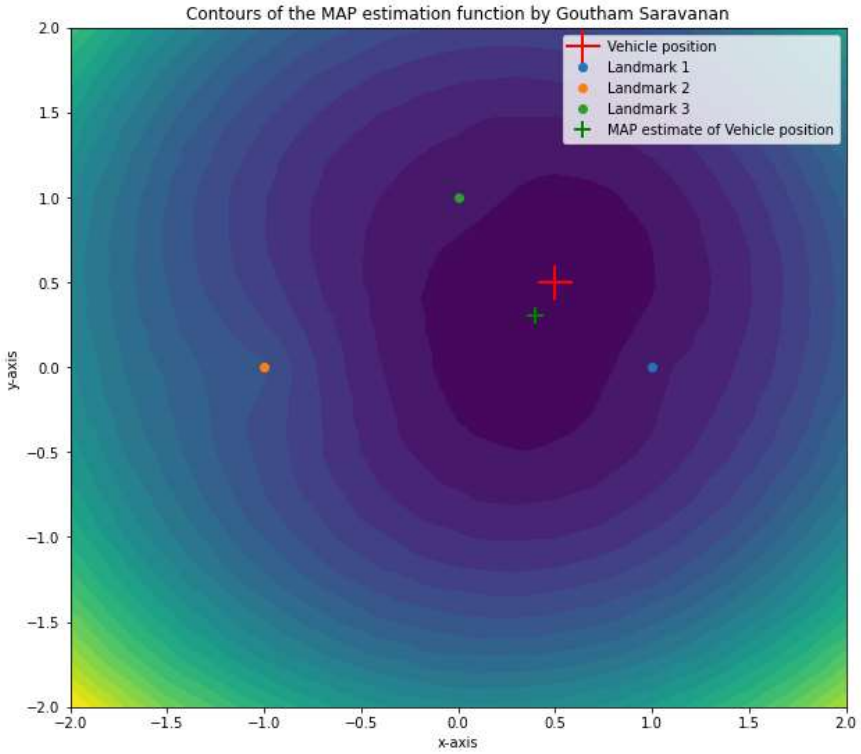
Eq. (G) is the simplified objective function //.



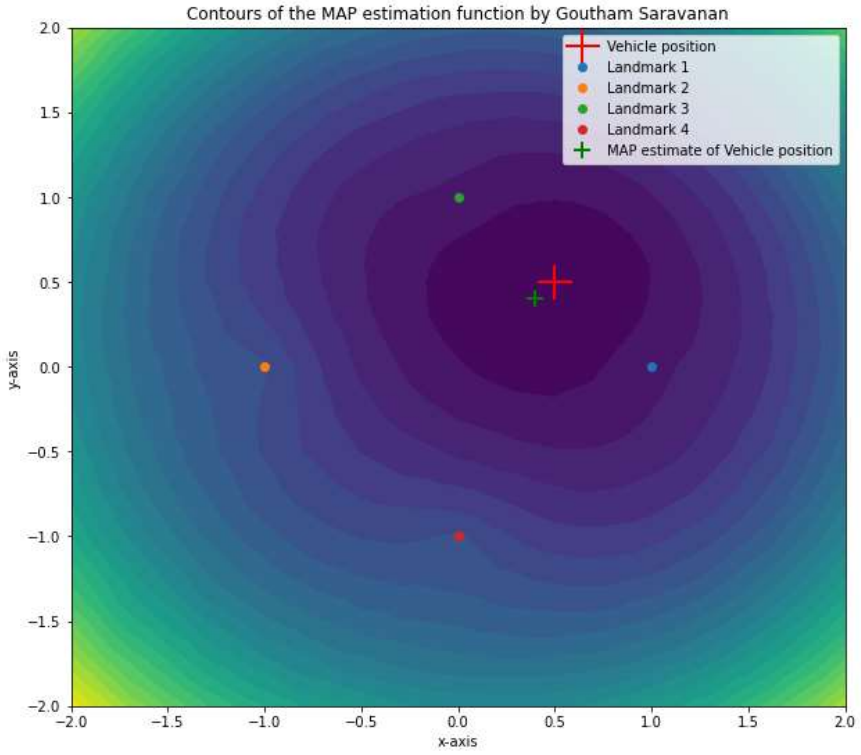
K=1 ; Sigma_x = 0.20 & Sigma_y = 0.20 (Set equal)



K=2 ; Sigma_x = 0.20 & Sigma_y = 0.20 (Set equal)



K=3 ; Sigma_x = 0.20 & Sigma_y = 0.20 (Set equal)



K=4 ; Sigma_x = 0.20 & Sigma_y = 0.20 (Set equal)

The code explanation: (Pasted below this page.)

Here, the contour is the main criteria that which the algorithm divides into the numerical points with its respective objective function. The contour is plotted based on the MAP estimate at its each and every point. The standard deviation of the i^{th} point and x and y axis are considered that they should balance the prior and likelihood in the function.

Overall, the MAP estimates are evaluated, the point which shows the minimum value of the objective function is considered to be the vehicle position given for the MAP estimator function.

MAP estimator explanation:

When the K=1, the vehicle position is near the centre of the contour as it shows the important role of the prior value.

As if the value of K increases say from 1 to 4 (given), we can also encounter that the estimated vehicle position is somewhat near the true position. This is due to the likelihood factor of the objective function influences due to the increasing number for landmarks.

Thus, I conclude when the value of the K increases the estimated vehicle position tends to move near the true position.

Python Code : //change the k value to 1,2,3,4 @ the line 16 each time.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal
from scipy.linalg import sqrtm
import math
x_points = np.arange(-2, 2.05, 0.1)
y_points = np.arange(-2, 2.05, 0.1)
x_mesh, y_mesh = np.meshgrid(x_points, y_points)
xy_map = np.zeros((len(x_mesh), len(y_mesh)))
landmark = np.zeros((2,4))
check = 1
x_t = 0.5
y_t = 0.5
land_x = [1, -1, 0, 0]
land_y = [0, 0, 1, -1]
K = 1
sig_i = 0.1
sig_x = 0.20
sig_y = 0.20
def cal_dti(x_t_d, y_t_d, x_i_d, y_i_d):
    dti = math.sqrt((x_t_d - x_i_d)**2 + (y_t_d - y_i_d)**2)
    return dti
for i in range(len(x_mesh)):
    for j in range(len(y_mesh)):
        likelihood = 0
        for q in range(K):
```

```
            n_i = np.random.normal(0, sig_i**2)

            r_i = cal_dti(x_t, y_t, land_x[q], land_y[q]) + n_i

            likelihood = likelihood + ((r_i - cal_dti(land_x[q], land_y[q], x_mesh[i,j],
y_mesh[i,j]))**2)/(sig_i**2)

        prior = (x_mesh[i,j]**2)/(sig_x**2) + (y_mesh[i,j]**2)/(sig_y**2)

        xy_map[i, j] = likelihood + prior

        if check == 1:

            mini = xy_map[i,j]

            check = 2

        if xy_map[i, j] < mini:

            mini = xy_map[i,j]

            mini_x = x_mesh[i,j]

            mini_y = y_mesh[i,j]

fig = plt.figure(figsize=(10, 9))

ax = fig.add_subplot(1, 1, 1)

cs = ax.contourf(x_mesh, y_mesh, xy_map, levels = 30)

ax.plot(x_t, y_t, 'r+', markeredgewidth = 2, markersize = 25, label = 'Vehicle position')

for q in range(K):

    ax.plot(land_x[q], land_y[q], 'o', label = 'Landmark {}'.format(q+1))

ax.plot(mini_x, mini_y, 'g+', markeredgewidth = 2, markersize = 12, label = 'MAP estimate of Vehicle
position')

plt.xlabel('x-axis')

plt.ylabel('y-axis')

plt.title('Contours of the MAP estimation function by Goutham Saravanan')

ax.legend()

plt.show()
```

Q3:

Given,

$$\lambda(\alpha_i|w_j) = \begin{cases} 0, & i = j \\ \lambda_r, & i = c+1 \\ \lambda_s & \text{otherwise} \end{cases}$$

From the given conditions, the definition of risk is expected loss,

$$R(\alpha_i|x) = \sum_{i=1,2..c}^{c} \lambda(\alpha_i|w_j) P(w_j|x)$$

$$R(\alpha_{c+1}|x) = \lambda_r \quad (\text{Given})$$

For the 3rd condition to be satisfied, it can be simplified.

$$R(\alpha_i|x) = \lambda_s \sum_{j=1}^{c} \lambda(\alpha_i|w_j) P(w_j|x)$$

$$= \lambda_s (1 - P(w_i|x))$$

As per the given question, it is mentioned to show the minimum risk between the posterior probabilities.

∴ We opt to choose smallest risk

$$\lambda_r < \lambda_s \ (1 - P(w_i|x)) \quad \text{for all } i = 1, 2 \ldots c$$

Also,

$$1 - \frac{\lambda_r}{\lambda_s} > P(w_i|x) \quad \text{for all } i = 1, 2, \ldots c$$

But, most of the cases we need to satisfy the posterior probabilities more than the $(1 - \frac{\lambda_r}{\lambda_s})$ LHS. Or else eliminate it.

Alternatively, in terms of i.

$$\lambda_s (1 - P(w_i|x)) < \lambda_s (1 - P(w_j|x)) \quad \text{for all } j = 1, 2 \ldots c, \ j \neq i$$

Then,

$$P(w_i|x) > P(w_j|x) \quad \text{for all } j \neq i$$

Concluding, that selecting the class with largest posteriori probability as it should be eliminated only if

$$P(w_i|x) < 1 - \frac{\lambda_r}{\lambda_s} \quad \text{for all } i = 1, 2 \ldots c \qquad (A)$$

If $\lambda_r = 0$, there is no influence on the result as it shows no loss & can be eliminated.

If $\lambda_r > \lambda_s$, the loss of rejecting could be more than compared to the loss of misclassifying. So it should never eliminated / reject.

As the value of λ_r is greater than λ_s, then the eq. (A') would be $1 - \frac{\lambda_r}{\lambda_s} < 0$. & turn's out to be false condition for the eq (A').

Appendix :

ForQ1:Python code

```
import numpy as np
import scipy.stats
import random
import matplotlib.pyplot as plt
import sys
from sklearn.mixture import GaussianMixture
from scipy.optimize import minimize
from matplotlib.colors import LogNorm
np.set_printoptions(suppress=True)

def calc_pxl(data, mean, cov):

    return scipy.stats.multivariate_normal.pdf(data, mean=mean,
cov=cov)

def calc_prob_threshs(sample_type, log_score, log_thresh_range):

    tps, tns, fps, fns, fs = [], [], [], [], []

    num_samples = sample_type[0]
    N0, N1 = sample_type[1]
    data_wt_labels = sample_type[2]
    labels = data_wt_labels[2,:]

    for log_thresh in log_thresh_range:

        tp, tn, fp, fn, f = calc_prob_thresh(log_score,
log_thresh, labels, N0, N1)
        tps.append(tp); fps.append(fp)
        tns.append(tn); fns.append(fn)
        fs.append(f)

    tps = np.array(tps); tns = np.array(tns)
    fps = np.array(fps); fns = np.array(fns)
    fs = np.array(fs)

    sample_type[3] = [tps, tns, fps, fns, fs]
```

```
    return sample_type

def calc_prob_thresh(log_score, log_thresh, labels, N0,N1):
```

```
decisions = (log_score>log_thresh).astype('int')
#print('decisions ',decisions)

    tp = np.sum(np.multiply(labels == 1,
 decisions==1).astype('int'))/N1
    fp = np.sum(np.multiply(labels == 0,
 decisions==1).astype('int'))/N0
    tn = np.sum(np.multiply(labels == 0,
 decisions==0).astype('int'))/N0
    fn = np.sum(np.multiply(labels == 1,
 decisions==0).astype('int'))/N1
    f = (fp*N0 + fn*N1)/(N0 +N1)

    return tp, tn, fp, fn, f

 def erm(sample_type, means, covs):

    #data_wt_labels (3, N)
    print('***** erm *****')
    m0, m1 = means
    C0, C1 = covs

    data_wt_labels = sample_type[2]
    pts = data_wt_labels[:2,:].T ##(N, 2)
    labels = data_wt_labels[2,:]

    px0_0 = scipy.stats.multivariate_normal.pdf(pts,
 mean=m0[0,:], cov=C0[0,:,:])
    px0_1 = scipy.stats.multivariate_normal.pdf(pts,
 mean=m0[1,:], cov=C0[1,:,:])

    px0 = w1*px0_0 + w2*px0_1 ##(N, 1)
    px1 = scipy.stats.multivariate_normal.pdf(pts, mean=m1,
 cov=C1) ##(N, 1)

    score = np.divide(px1, px0)
    log_score = np.log(score)
    sort_log_score = np.sort(log_score)  ##(N, 1)

    eps = 1e-3
    log_thresh_range = np.append(sort_log_score[0] - eps,
 sort_log_score + eps)
```

```
    sample_type = calc_prob_threshs(sample_type, log_score,
log_thresh_range)

    # theoretical
    log_thresh_t =np.log(pL[0]/pL[1])
    N0, N1 = sample_type[1]
    tp_t, tn_t, fp_t, fn_t, f_t = calc_prob_thresh(log_score,
log_thresh_t, labels, N0, N1)

    # min PE thresh from data
    tps, tns, fps, fns, fs = sample_type[3]
    min_poe = np.min(fs)
    min_poe_ids = np.where(fs==min_poe)[0]

    # get closest thresh to theoretical
    min_dist, min_id = sys.maxsize, 0
    for id in min_poe_ids:
        dist = log_thresh_range[id] - log_thresh_t
        if dist<min_dist:
            min_dist = dist
            min_id = id

    print('min_poe_t ',f_t)
    print('min_poe_a ',fs[min_id])
    print('min_poe_thresh ',np.exp(log_thresh_range[min_id]))
    print('thresh_t ',np.exp(log_thresh_t))

    #ROC curve
    plt.plot(fps, tps, label='ROC Curve')
    plt.plot(fps[min_id], tps[min_id], 'ro', label='Estimated
Minimum Error')
    plt.plot(fp_t, tp_t, 'g+', label='Theoretical Minimum Error')
    plt.title('Minimum Expected Risk ROC Curve')
    plt.xlabel('Prob. False positives')
    plt.ylabel('Prob. True positives')
    plt.legend()
    plt.show()

    # Probability of Error
    plt.plot(log_thresh_range, fs, label='Probability of Error')
    plt.plot(log_thresh_range[min_id], fs[min_id], 'ro',
label='Estimated Minimum Error threshold')
```

```
    plt.plot(log_thresh_t, f_t, 'g+', label='Theoretical
Threshold')
    plt.title('Probability of Error vs log_threshold')
    plt.xlabel('log_threshold')
    plt.ylabel('Probability of Error')
    plt.legend()
    plt.show()

    # Decision boundary
    log_score = np.log(score)
    decisions = (log_score>log_thresh_t).astype('int')
    pts = pts.T
    plot_boundary(pts, labels,decisions)
    hgrid =
np.linspace(np.floor(min(pts[0,:])),np.ceil(max(pts[0,:])),100)
    vgrid =
np.linspace(np.floor(min(pts[1,:])),np.ceil(max(pts[1,:])),100)
    dsg = np.zeros((100,100))
    mat = np.array(np.meshgrid(hgrid, vgrid))

    for i in range(100):
        for j in range(100):
            px0_0 =
scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j],
mat[1][i][j]]), mean=m0[0,:], cov=C0[0,:,:])
            px0_1 =
scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j],
mat[1][i][j]]), mean=m0[1,:], cov=C0[1,:,:])
            px0 = w1*px0_0 + w2*px0_1 ##(N, 1)
            px1 =
scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j],
mat[1][i][j]]), mean=m1, cov=C1) ##(N, 1)
            dsg[i][j] = np.log(px0) - np.log(px1) - np.log(pL[0]/
pL[1])

    plt.contour(mat[0], mat[1],dsg)
    plt.show()

def split_data(data_wt_labels):

    l0_ids = np.where(data_wt_labels[2,:]==0)[0]
    l1_ids = np.where(data_wt_labels[2,:]==1)[0]
```

```
    data0 =data_wt_labels[:,l0_ids]
    data1 =data_wt_labels[:,l1_ids]

    return data0, data1

def print_gmm_params(gmm_l0, gmm_l1):

    print('GMM params L0 ',gmm_l0.get_params())
    print('GMM params L1 ',gmm_l1.get_params())

    if gmm_l0.converged_:print('Label 0 converged')
    else:print('Label 0 not converged')

    if gmm_l1.converged_:print('Label 1 converged')
    else:print('Label 1 not converged')

    print('Label 0 weights ',gmm_l0.weights_,
gmm_l0.weights_.shape)
    print('Label 1 weights ',gmm_l1.weights_,
gmm_l1.weights_.shape)

    print('Label 0 means ',gmm_l0.means_.shape)
    print('Label 0 covariances ',gmm_l0.covariances_.shape)

    print('Label 1 means ',gmm_l1.means_.shape)
    print('Label 1 covariances ',gmm_l1.covariances_.shape)


def mle_gmm(train_sample_type, val_sample_type):

    data_wt_labels = train_sample_type[2]
    data0, data1 = split_data(data_wt_labels)
    data0, data1 = data0[:2, :].T, data1[:2, :].T

    gmm_l0 = GaussianMixture(2, covariance_type='full',
                    random_state=0).fit(data0)

    gmm_l1 = GaussianMixture(1, covariance_type='full',
                    random_state=0).fit(data1)

    #print_gmm_params(gmm_l0, gmm_l1)
```

```
    m01 = gmm_l0.means_[0,:]
    m02 = gmm_l0.means_[1,:]
    C01 = gmm_l0.covariances_[0,:]
    C02 = gmm_l0.covariances_[1,:]
    gmm_weights0 = gmm_l0.weights_

    w1 = gmm_weights0[0]; w2 = gmm_weights0[1]

    m1 = gmm_l1.means_[0,:]
    C1 = gmm_l1.covariances_[0,:]
    gmm_weights1 = gmm_l1.weights_

    print('C01: ', C01)
    print('C02: ', C02)
    print('C1: ', C1)

    print('m01: ', m01)
    print('m02: ', m02)
    print('m1: ', m1)

    print('w1 ',w1)
    print('w2 ',w2)

    data_wt_labels = val_sample_type[2]
    pts = data_wt_labels[:2,:].T ##(N, 2)
    labels = data_wt_labels[2,:]

    px0_0 = scipy.stats.multivariate_normal.pdf(pts, mean=m01,
cov=C01)
    px0_1 = scipy.stats.multivariate_normal.pdf(pts, mean=m02,
cov=C02)

    px0 = w1*px0_0 + w2*px0_1 ##(N, 1)
    px1 = scipy.stats.multivariate_normal.pdf(pts, mean=m1,
cov=C1) ##(N, 1)

    score = np.divide(px1, px0)
    log_score = np.log(score)
    sort_log_score = np.sort(log_score)  ##(N, 1)

    eps = 1e-3
```

```
    log_thresh_range = np.append(sort_log_score[0] - eps,
sort_log_score + eps)
    val_sample_type = calc_prob_threshs(val_sample_type,
log_score, log_thresh_range)

    # theoretical
    log_thresh_t =np.log(pL[0]/pL[1])
    N0, N1 = val_sample_type[1]
    tp_t, tn_t, fp_t, fn_t, f_t = calc_prob_thresh(log_score,
log_thresh_t, labels, N0, N1)

    # min PE thresh from data
    tps, tns, fps, fns, fs = val_sample_type[3]
    min_poe = np.min(fs)
    min_poe_ids = np.where(fs==min_poe)[0]

    # get closest thresh to theoretical
    min_dist, min_id = sys.maxsize, 0
    for id in min_poe_ids:
        dist = log_thresh_range[id] - log_thresh_t
        if dist<min_dist:
            min_dist = dist
            min_id = id

    print('min_poe_t ',f_t)
    print('min_poe_a ',fs[min_id])
    print('min_poe_thresh ',np.exp(log_thresh_range[min_id]))
    print('thresh_t ',np.exp(log_thresh_t))

    #ROC curve
    plt.plot(fps, tps, label='ROC Curve')
    plt.plot(fps[min_id], tps[min_id], 'ro', label='Estimated
Minimum Error Threshold')
    plt.plot(fp_t, tp_t, 'g+', label='Theoretical Minimum Error')
    plt.title('ROC Curve')
    plt.xlabel('Prob. False positives')
    plt.ylabel('Prob. True positives')
    plt.legend()
    plt.show()

    # Probability of Error
    plt.plot(log_thresh_range, fs, label='Probability of Error')
```

```
    plt.plot(log_thresh_range[min_id], fs[min_id], 'ro',
label='Estimated Minimum Error threshold')
    plt.plot(log_thresh_t, f_t, 'g+', label='Theoretical
Threshold')
    plt.title('Probability of Error vs log_threshold')
    plt.xlabel('log_threshold')
    plt.ylabel('Probability of Error')
    plt.legend()
    plt.show()

    # GMM contour for Class 0
    data_wt_labels =train_sample_type[2]
    pts = data_wt_labels[:2,:] ##(2, N)
    hgrid =
np.linspace(np.floor(min(pts[0,:])),np.ceil(max(pts[0,:])),100)
    vgrid =
np.linspace(np.floor(min(pts[1,:])),np.ceil(max(pts[1,:])),100)
    dsg = np.zeros((100,100))
    mat = np.array(np.meshgrid(hgrid, vgrid))

    for i in range(100):
        for j in range(100):
            px0_0 =
scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j],
mat[1][i][j]]), mean=m0[0,:], cov=C0[0,:,:])
            px0_1 =
scipy.stats.multivariate_normal.pdf(np.array([mat[0][i][j],
mat[1][i][j]]), mean=m0[1,:], cov=C0[1,:,:])
            dsg[i][j] = w1*px0_0 + w2*px0_1 ##(N, 1)

    CS = plt.contour(mat[0], mat[1], dsg)
    CB = plt.colorbar(CS, shrink=0.8, extend='both')
    plt.scatter(pts[0,:], pts[1,:], .8)
    plt.title('Contour Plot for Class 0 Estimated GMM for ' +
str(pts.shape[1]) + 'samples')
    plt.xlabel("X1")
    plt.ylabel("X2")
    plt.show()

def calc_cost(x, data, labels):

    w = x
```

```
    h = 1 / (1+ np.exp(-(np.dot(w.T,data)))) ##(N, )
    loss = labels * np.log(h) + (1 - labels) * np.log(1 - h)
##(N, )

    sum = np.sum(loss)
    scale = -(1.0 /data.shape[1])
    cost = scale * sum

    return cost

def predict(w, data, thresh=0.5):

    h = 1 / (1+ np.exp(-(np.dot(w.T,data))))

    h[h>=thresh] = 1
    h[h<thresh] = 0

    return h

def mle_opt_lin(train_sample_type, test_sample_type):

    train_data_wt_labels = train_sample_type[2]
    train_data = train_data_wt_labels[:2, :]
    train_labels = train_data_wt_labels[2,:]
    train_ones =
np.ones(train_data_wt_labels.shape[1]).reshape((1, -1))
    train_data = np.concatenate((train_ones, train_data), axis=0)
##(3, N)

    print('training.. ')
    w_init = np.zeros((3, 1), dtype='float')
    result = minimize(calc_cost, w_init, args=(train_data,
train_labels))
    w_trained = result.x
    print('training completed!')

    print('w_trained ',w_trained)
    test_data_wt_labels = test_sample_type[2]
    test_data = test_data_wt_labels[:2, :]
    test_labels = test_data_wt_labels[2,:]
    test_ones = np.ones(test_data_wt_labels.shape[1]).reshape((1,
```

```
-1))
    test_data = np.concatenate((test_ones, test_data), axis=0)
##(3, N)

    decisions = predict(w_trained, test_data)
    acc = calc_poe(decisions, test_labels)
    print('acc ',acc)

    plot_boundary(test_data_wt_labels[:2, :], test_labels,
decisions)
    mat = get_mesh_grid(test_data_wt_labels[:2, :])
    boundary = np.zeros((100, 100))
    for i in range(100):
        for j in range(100):
            x1 = mat[0][i][j]
            x2 = mat[1][i][j]
            z = np.c_[1, x1, x2].T
            boundary[i][j] = np.sum(np.dot(w_trained.T, z))
    plt.contour(mat[0], mat[1], boundary, levels = [0])
    plt.show()

def gen_quad_data(data_wt_labels):

    num_samples =data_wt_labels.shape[1]
    input_data = data_wt_labels[:2, :] # (x1, x2)
    data = np.zeros((6, num_samples), dtype='float') # (1, x1,
x2, x1**2, x1x2, x2**2)

    data[0] = np.ones(num_samples).reshape((1, -1)) # 1
    data[1:3] = input_data #x1, x2
    data[3] = np.square(input_data[0, :]) # x1**2
    data[4] = np.multiply(input_data[0, :], input_data[1, :]) #
x1x2
    data[5] = np.square(input_data[1, :]) # x2**2

    return data

def mle_opt_quad(train_sample_type, test_sample_type):

    train_data_wt_labels = train_sample_type[2]
    train_labels = train_data_wt_labels[2,:]
    train_data = gen_quad_data(train_data_wt_labels)
```

```
    print('training.. ')
    w_init = np.zeros((6, 1), dtype='float')
    result = minimize(calc_cost, w_init, args=(train_data,
train_labels))
    w_trained = result.x
    print('training completed!')

    print('w_trained ',w_trained)
    test_data_wt_labels = test_sample_type[2]
    test_data = gen_quad_data(test_data_wt_labels)
    test_labels = test_data_wt_labels[2,:]

    decisions = predict(w_trained, test_data)
    acc = calc_poe(decisions, test_labels)
    print('acc ',acc)

    plot_boundary(test_data_wt_labels[:2, :], test_labels,
decisions)
    mat = get_mesh_grid(test_data_wt_labels[:2, :])
    boundary = np.zeros((100, 100))
    for i in range(100):
        for j in range(100):
            x1 = mat[0][i][j]
            x2 = mat[1][i][j]
            z = np.c_[1, x1, x2, x1**2, x1*x2, x2**2].T
            boundary[i][j] = np.sum(np.dot(w_trained.T, z))
    plt.contour(mat[0], mat[1], boundary, levels = [0])
    plt.show()

def calc_poe(decisions, labels):

    N0 = np.sum((labels == 0).astype('int'))
    N1 = np.sum((labels == 1).astype('int'))

    tp = np.sum(np.multiply(labels == 1,
decisions==1).astype('int'))/N1
    fp = np.sum(np.multiply(labels == 0,
decisions==1).astype('int'))/N0
    tn = np.sum(np.multiply(labels == 0,
decisions==0).astype('int'))/N0
    fn = np.sum(np.multiply(labels == 1,
```

```
decisions==0).astype('int'))/N1
    f = (fp*N0 + fn*N1)/(N0 +N1)

    return (tp*N1 + tn*N0)/(N0 + N1)

def mle_opt(train_sample_type, test_sample_type, part=1):

    if part==1:
        mle_opt_lin(train_sample_type, test_sample_type)
    else:
        mle_opt_quad(train_sample_type, test_sample_type)

def plot_boundary(data, labels, decisions):

    tp = np.multiply(labels == 1, decisions == 1).astype('int')
    tn = np.multiply(labels == 0, decisions == 0).astype('int')
    fp = np.multiply(labels == 0, decisions == 1).astype('int')
    fn = np.multiply(labels == 1, decisions == 0).astype('int')

    tp_ids = np.where(tp == 1)[0]
    tn_ids = np.where(tn == 1)[0]
    fp_ids = np.where(fp == 1)[0]
    fn_ids = np.where(fn == 1)[0]

    plt.plot(data[0,  tn_ids], data[1, tn_ids], '+', color ='g',
markersize = 6)
     plt.plot(data[0, tp_ids], data[1, tp_ids], '.', color ='g',
markersize = 6)
     plt.plot(data[0, fp_ids], data[1, fp_ids], '+', color ='r',
markersize = 6)
     plt.plot(data[0, fn_ids], data[1, fn_ids], '.', color ='r',
markersize = 6)

    plt.legend(["class 0 correctly classified", 'class 1
correctly classified','class 0 wrongly classified', 'class 1
wrongly classified'])
    plt.title('Prediction overlapped with decision boundary')
    plt.xlabel("X1")
    plt.ylabel("X2")

def get_mesh_grid(data, num_grid=100):
```

```
    hgrid = np.linspace(np.floor(min(data[0,:])),
np.ceil(max(data[0,:])), num_grid)
    vgrid = np.linspace(np.floor(min(data[1,:])),
np.ceil(max(data[1,:])), num_grid)
    mat = np.array(np.meshgrid(hgrid, vgrid))

    return mat

def plot_dist(data, label_names):

    tname, xname, yname = label_names

    print('***** plot *****')
    data0, data1 = split_data(data)

    plt.scatter(data0[0, :], data0[1, :], s=5, color = 'red',
label = 'class 0',marker='*')
    plt.scatter(data1[0, :], data1[1, :], s=5, color = 'blue',
label = 'class 1', marker='*')

    plt.title(tname)
    plt.xlabel(xname)
    plt.ylabel(yname)
    plt.legend()
    plt.show()

def generate_data_pxgl(prior, means, covs, num_samples):

    m0, m1 = means
    C0, C1 = covs

    N0 = int(prior[0]*num_samples)
    N1 = num_samples - N0
    print('N0, N1 ',N0, N1)

    # generate L0
    N00 = int(w1*N0)
    N01 = N0 - N00
    wt_dist = [0]*N00 + [1]*N01
    for i in range(10):
        random.shuffle(wt_dist)
    wt_dist = np.array(wt_dist)
```

```
    dist0 = np.random.multivariate_normal(m0[0,:], C0[0,:,:],
N0).T
    dist1 = np.random.multivariate_normal(m0[1,:], C0[1,:,:],
N0).T
    pxgl0 = np.multiply(1-wt_dist, dist0) + np.multiply(wt_dist,
dist1)

    # generate L1
    pxgl1 = np.random.multivariate_normal(m1, C1, N1).T

    ## combine data and label
    labels = [0]*N0 + [1]*N1
    labels = np.reshape(labels, (1, -1))

    pxgl = np.concatenate((pxgl0, pxgl1), axis=1)
    data = np.concatenate((pxgl, labels), axis=0)

    return data, N0, N1

def generate_data_pxgl_samples(samples_type):

    for i, key in enumerate(samples_type.keys()):

        sample_type = samples_type[key]
        num_samples =int(sample_type[0][0])

        data_wt_labels, N0, N1 = generate_data_pxgl(pL, [m0, m1],
[C0, C1], num_samples)

        sample_type[1] = [N0, N1]
        sample_type[2] =data_wt_labels

        label_names = ["True label distribution for " +
str(num_samples) + " for two classes", "x1","x2"]
        plot_dist(data_wt_labels, label_names)

    return samples_type

if_name == "_main__":

    dim = 2
```

```
    #priors
    pL = [0.6, 0.4]

    #means
    m0 = np.array([[5, 0], [0, 4]])
    m1 = [3, 2]

    #covariance
    C0 = np.zeros((2,2,2), dtype=int)
    C0[0,:,:] = np.array([[4, 0], [0, 2]])
    C0[1,:,:] = np.array([[1, 0], [0, 3]])
    C1 = np.array([[2, 0], [0, 2]])

    ## gaus weight
    w1 = 0.5; w2 = 0.5

    # data
    ## num_samples, [N0, N1], data_wt_labels, [tps, tns, fps,
fns, fs]
    samples_type = {
        'D100': [[100], [], [], []],
        'D1k': [[1000], [], [], []],
        'D10k': [[10000], [], [], []],
        'D20k': [[20000], [], [], []],
    }

    ## generate data for all samples
    samples_type = generate_data_pxgl_samples(samples_type)

    erm_ = 0
    gmm_ = 0
    opt_ = 1

    ## erm
    if erm_:
        print('Part1')
        erm(samples_type['D20k'], [m0, m1], [C0, C1])

    ## mle_gmm
    if gmm_:
        print('Part2')
```

```
        for i, key in enumerate(list(samples_type.keys())[:-1]):

            print('************************************')
            print('train: ',key,' val: D20k')
            mle_gmm(samples_type[key], samples_type['D20k'])
            print('************************************')

            # if i==0:break

    if opt_:
        print('Part3')
        for i, key in enumerate(list(samples_type.keys())[:-1]):

            print('************************************')
            print('train: ',key,' val: D20k')
            mle_opt(samples_type[key],  samples_type['D20k'],
part=1)
            mle_opt(samples_type[key], samples_type['D20k'],
part=2)
            print('************************************')
```

Citations :

1.Github.com/nandavyk

2.Maximum Likelihood Estimation of Gaussian Parameters (jrmeyer.github.io)

3.sklearn.mixture.GaussianMixture — scikit-learn 1.0.1 documentation

4.Maximum Likelihood Estimation Explained - Normal Distribution | by Marissa Eppes | Towards Data Science